What if *each individual page* was this kind of mish-mash? What if every page read like this:

> You exit the passageway into a large cavern. Unless you came from page 59, in which case you fall down the sinkhole into a large cavern. A huge troll, or possibly a badger (if you already visited Queen Pelican), blocks your path. Unless you threw a button down the wishing well on page 8, in which case there nothing blocking your way. The [troll or badger or nothing at all] does not look happy to see you.
>
> If you came here from chapter 7 (the Pool of Time), go back to the top of the page and read it again, only imagine you are watching the events happen to someone else.
>
> If you already received the invisibility cloak from the aged lighthouse-keeper, and you want to use it now, go to page 67. Otherwise, forget you read anything about an invisibility cloak.
>
> If you are facing a troll (see above), and you choose to run away, turn to page 84.
>
> If you are facing a badger (see above), and you choose to run away, turn to page 93…

Not the most compelling narrative, is it? The story asks you to carry so much mental baggage for it that just getting through a page is exhausting.

Code as narrative

What does this have to do with software? Well, code can tell a story as well. It might not be a tale of high adventure and intrigue. But it's a story nonetheless; one about

a problem that needed to be solved, and the path the developer(s) chose to accomplish that task.

A single method is like a page in that story. And unfortunately, a lot of methods are just as convoluted, equivocal, and confusing as that made-up page above.

In this book, we'll take a look at many examples of the kind of code that unnecessarily obscures the storyline of a method. We'll also explore a number of techniques for minimizing distractions and writing methods that straightforwardly convey their intent.

The four parts of a method

I believe that if we take a look at any given line of code in a method, we can nearly always categorize it as serving one of the following roles:

1. Collecting input
2. Performing work
3. Delivering output
4. Handling failures

(There are two other categories that sometimes appear: "diagnostics", and "cleanup". But these are less common.)

Let's test this assertion. Here's a method taken from the MetricFu project.

```
def location(item, value)
  sub_table = get_sub_table(item, value)
  if(sub_table.length==0)
    raise MetricFu::AnalysisError, "The #{item.to_s}
'#{value.to_s}' "\
    "does not have any rows in the analysis table"
  else
    first_row = sub_table[0]
    case item
    when :class
      MetricFu::Location.get(first_row.file_path,
first_row.class_name, nil)
    when :method
      MetricFu::Location.get(first_row.file_path,
first_row.class_name, first_row.method_name)
    when :file
      MetricFu::Location.get(first_row.file_path, nil, nil)
    else
      raise ArgumentError, "Item must be :class, :method, or :file"
    end
  end
end
```

Don't worry too much right now about what this method is supposed to do. Instead, let's see if we can break the method down according to our four categories.

First, it gathers some input:

```
sub_table = get_sub_table(item, value)
```

Immediately, there is a digression to deal with an error case, when `sub_table` has no data.

```
    if(sub_table.length==0)
      raise MetricFu::AnalysisError, "The #{item.to_s} '#{value.to_s}'
    "\
      "does not have any rows in the analysis table"
```

Then it returns briefly to input gathering:

```
    else
      first_row = sub_table[0]
```

Before launching into the "meat" of the method.

```
    when :class
      MetricFu::Location.get(first_row.file_path, first_row.class_name,
    nil)
    when :method
      MetricFu::Location.get(first_row.file_path, first_row.class_name,
    first_row.method_name)
    when :file
      MetricFu::Location.get(first_row.file_path, nil, nil)
```

The method ends with code dedicated to another failure mode:

```
    else
        raise ArgumentError, "Item must be :class, :method, or :file"
      end
    end
```

Let's represent this breakdown visually, using different colors to represent the different parts of a method.

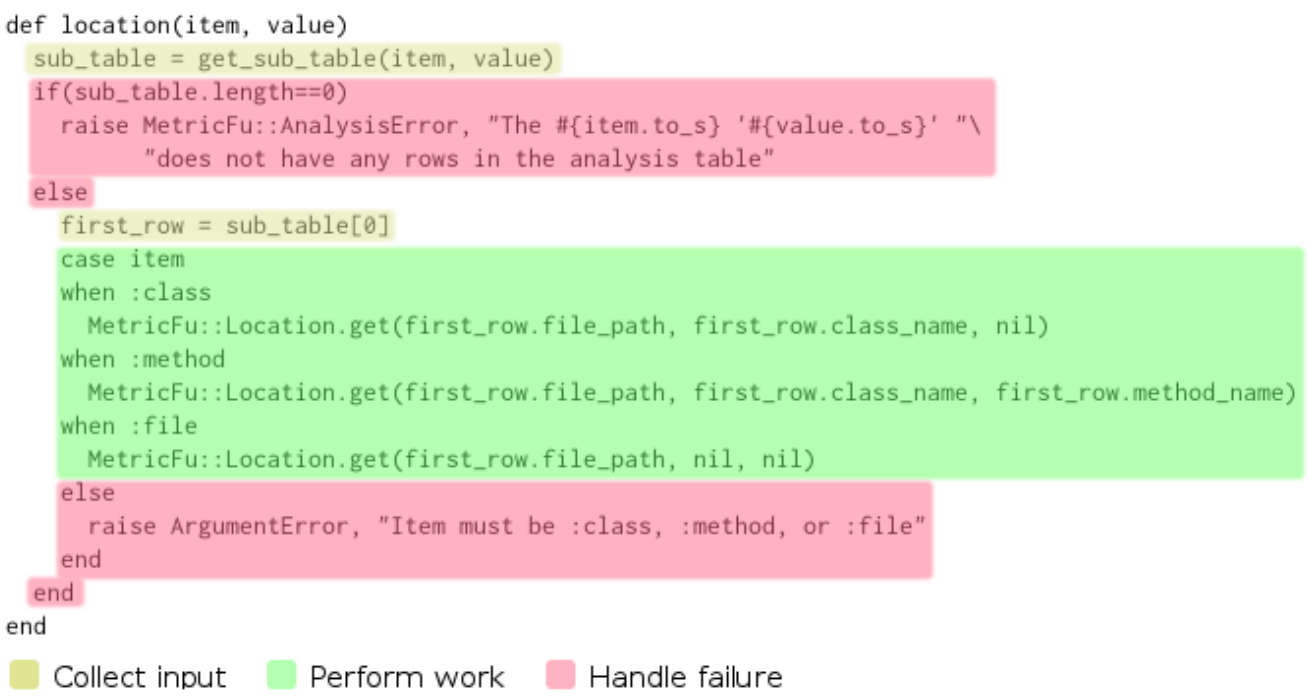

Figure 1: The #location method, annotated

This method has no lines dedicated to delivering output, so we haven't included that in the markup. Also, note that we mark up the top-level `else...end` delimiters as "handling failure". This is because they wouldn't exist without the preceding `if` block, which detects and deals with a failure case.

The point I want to draw your attention to in breaking down the method in this way is that the different parts of the method are mixed up. Some input is collected; then some error handling; then some more input collection; then work is done; and so on.

This is a defining characteristic of "un-confident", or as I think of it, "timid code": the haphazard mixing of the parts of a method. Just like the confused adventure story earlier, code like this puts an extra cognitive burden on the reader as they

unconsciously try to keep up with it. And because its responsibilities are disorganized, this kind of code is often difficult to refactor and rearrange without breaking it.

In my experience (and opinion), methods that tell a good story lump these four parts of a method together in distinct "stripes", rather than mixing them will-nilly. But not only that, they do it in the order I listed above: First, collect input. Then perform work. Then deliver output. Finally, handle failure, if necessary.

(By the way, we'll revisit this method again in the last section of the book, and refactor it to tell a better story.)

How this book is structured

This book is a patterns catalog at heart. The patterns here deal with what Steve McConnell calls "code construction" in his book Code Complete. They are "implementation patterns", to use Kent Beck's terminology. That means that unlike the patterns in books like Design Patterns or Patterns of Enterprise Application Architecture, most of these patterns are "little". They are not architectural. They deal primarily with how to structure individual methods. Some of them may seem more like idioms or stylistic guidelines than heavyweight patterns.

The material I present here is intended to help you write straightforward methods that follow this four-part narrative flow. I've broken it down into six parts:

- First, a discussion of writing methods in terms of *messages* and *roles*.
- Next, a chapter on "Performing Work". While it may seem out of order based on the "parts of a method" I laid out above, this chapter will equip you with a way of thinking through the design of your methods which will set the stage for the patterns to come.